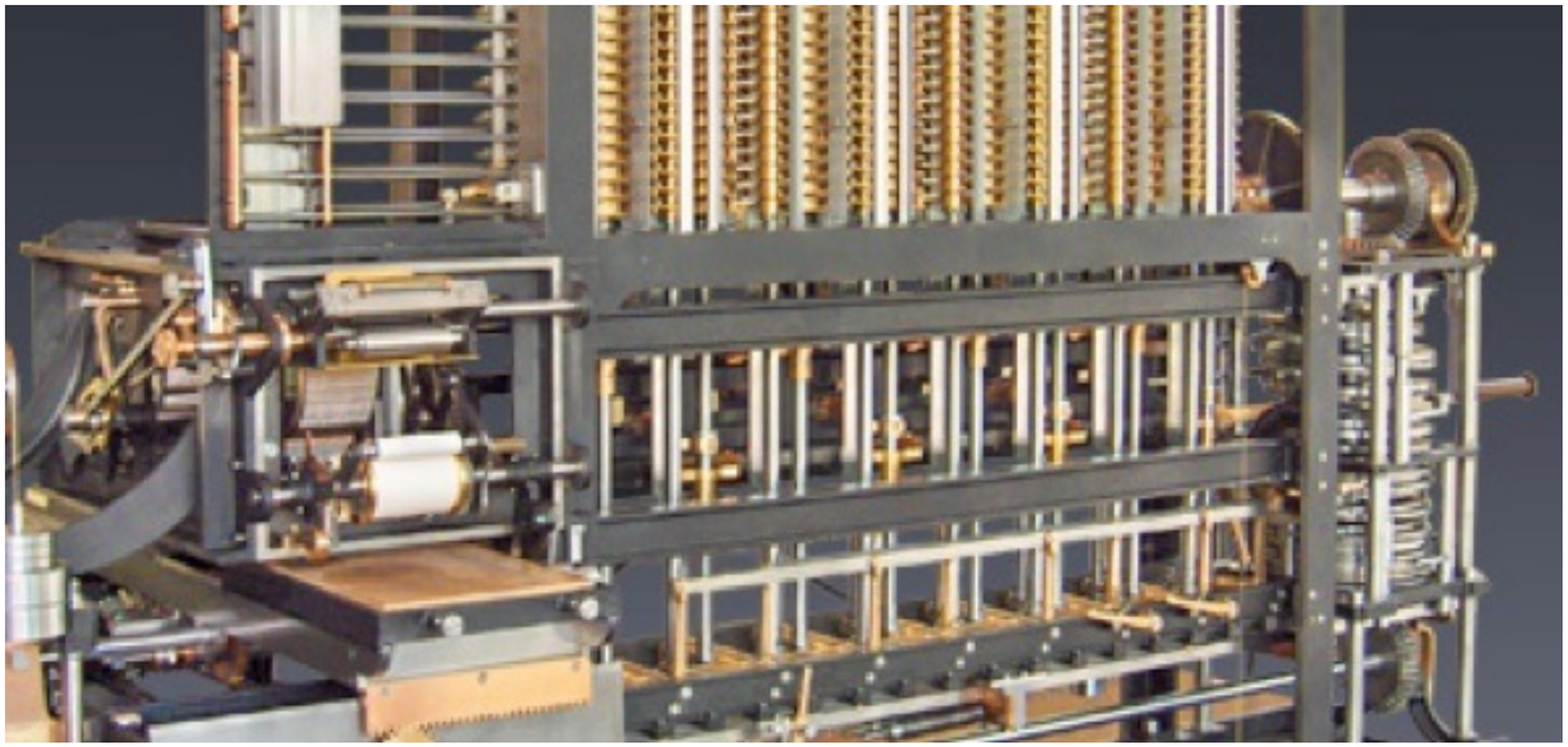
# UE Programmation C++

## Master Sciences pour l'Ingénieur

- Classes
  - Définition
  - Encapsulation
  - Accesseurs
  - Constructeurs
  - Destructeurs
- Exercices



2ème cours

Alice Cohen-Hadria

## Les classes: définition



- Un objet est constitué d'un ensemble de **données** de type quelconque et d'un ensemble de **fonctions** permettant de manipuler ces données.

- La définition d'un type d'objet s'appelle une **classe**. Les classes représentent le principal mécanisme par lequel il devient possible de définir de nouveaux types de données en C++, donc d'étendre les types de base fournis par le langage.

- Du point de vue de la syntaxe, la définition d'une classe ressemble beaucoup à celle d'une structure à ceci près que le mot réservé **class** remplace le mot struct.

- Certains champs ne sont pas des données mais des fonctions.

# Les classes: encapsulation



Dans cet exemple, la classe est séparée en deux parties :
l'une qui contient les données (x et y) et qui est déclarée **private**
L'autre qui contient des fonctions et qui est déclarée **public :**

```
class Complex {
private :
double x, y;                          // attributs

public :
  void init( double a, double b ) {    // méthode
    x = a;
    y = b;
}

void afficher() {
    cout << x << " + i*" << y;
}

};

void main() {
  Complex c;
  c.init( 2., 2. );                   // accès aux données par les méthodes
  c.afficher();
}
```

# Les classes: accès aux membres



Avec une classe définie ainsi :

```
class Complex {
private :
  double x, y;
public :
  void set_x ( double _x) { x = _x; };      // accesseurs...
  void set_y ( double _y) { x = _y; };
  double get_x() const { return x;};        // méthode constante
  double get_y() const { return y;};        // ...
};

void main() {
  Complex c;
  c.x = 2.;                                 // erreur à la compilation
  c.y = 2.;                                 // erreur à la compilation
  c.set_x( 2. );                            // OK
  cout << "partie réelle = " << c.get_x()  << endl;
}
```

Accès par pointeurs :

```
  Complex *pc;
  pc = new Complex;
  pc->set_x( 2.0 );
```

# Les classes: le pointeur *this*



Avec une classe définie ainsi :

```
class Complex {
private :
  double x, y;
public :
  void set_x ( double x) { this->x = x; };        // differentiation des attributs et paramètres
  void set_y ( double y) { (*this).y = y; };
  double get_x() const { return x;};
  double get_y() const { return y;};

  void afficher() {
    cout << this->x << " + i*" << this->y;
  }
```

Rien ne change :

```
void main() {
  Complex c;
  c.x = 2.;                                       // erreur à la compilation
  c.y = 2.;                                       // erreur à la compilation
  c.set_x( 2. );                                  // OK
  cout << "partie réelle = " << c.get_x()  << endl;
  c.afficher();
}
```

# Les classes: accès par références



Avec une classe définie ainsi :

```
class Complex {
private :
  double x, y;
public :
//  void set_x ( double x) { this->x = x; };        // méthode inutile
//  void set_y ( double y) { (*this).y = y; };      // méthode inutile
   double &get_x() { return x;};                        // accès en écriture
   double &get_y() { return y;};
   double get_x() const { return x;};                   // accès en lecture
   double get_y() const { return y;};
 }
```

Rien ne change :

```
void main() {
  Complex c;
  c.get_x() = 2.;                                  // affectation
  cout << "partie réelle = " << c.get_x()  << endl;  // lecture
}
```

# Constructeur d'une classe



Le constructeur d'une classe désigne une fonction membre particulière qui a pour rôle d'initialiser l'objet (typiquement donner des valeurs à ses attributs). La mémoire est déjà allouée pour l'objet. Le constructeur d'une classe à le même nom que la classe, n'a pas de type de retour et ne retourne aucune valeur (pas d'instruction return dans le corps d'un constructeur.

```
class Complex {
private :
  double x, y;
public :
  Complex( double a, double b ) {
    x = a;
    y = b; }
  void afficher() {
    cout << x << " + i*" << y;
  }
};
```

Le constructeur initialise les deux attributs x et y de la classe Complex.

# Constructeur d'une classe



Un constructeur est toujours appelé implicitement (lorsqu'un objet est créé) ou explicitement. Il peut être surchargé lorsque l'on veut mettre plusieurs signatures à la disposition de l'utilisateur. Exemple :

```
class Complex {
private :
  double x, y;
public :
  Complex( double a, double b ) { x = a; y = b; }
  Complex( double a ) { x = a; y = 0.; }
  Complex( ) { x = 0.; y = 0.; }
};
```

# Constructeur d'une classe



Un constructeur est toujours appelé implicitement (lorsqu'un objet est créé) ou explicitement. Il peut être surchargé lorsque l'on veut mettre plusieurs signatures à la disposition de l'utilisateur. Exemple :

```
class Complex {
private :
  double x, y;
public :
  Complex( double a, double b ) { x = a; y = b; }
  Complex( double a ) { x = a; y = 0.; }
  Complex( ) { x = 0.; y = 0.; }
};
```

En utilisant des paramètres par défaut les constructeurs peuvent être regroupés :

```
class Complex {
private :
  double x, y;
public :
  Complex( double a=0., double b=0. ) { x = a; y = b; }
};
```

## Appel des constructeurs

Les appels explicites des constructeurs peuvent se faire de trois façons différentes :

```
Complex c( 2., 2. );
Complex d = Complex( 2., 2. ); // Attention à ne pas confondre initialisation et affectation
Complex e = 2.;     // seulement dans le cas d'un constructeur à un seul paramètre
```

La dernière ligne est équivalente à :

```
Complex e( 2 );
```

Lors d'une création dynamique on écrira :

```
Complex *pc;
Complex pc = new Complex( 2., 2. );
```

Les constructeurs sont aussi appelés lors de la création d'objets temporaires. Dans l'exemple ci-dessous le constructeur est appelé une fois :

```
cout << Complex( 2., 2. ) << endl;
```

XXX cout non surchargé à ce stade

# Le constructeur par défaut



Ce constructeur est utilisé **sans paramètres**: soit il n'a pas de paramètres, soit ils sont initialisés par défaut. Il est appelé à chaque fois qu'un objet est créé **sans appel explicite au constructeur de la classe** :

```
Complex c;
Complex d[20];
Complex *pc = new Complex();
```

**Le travail du compilateur :**
Lorsqu'aucun constructeur par défaut n'est défini dans une classe, le compilateur génère un constructeur par défaut dont la caractéristique est de ne rien faire, hormis appeler les constructeurs par défaut de chacun des objets déclarés dans la classe.

Lorsqu'un constructeur par défaut est défini, le compilateur n'en génère pas un lui-même. Ainsi, l'appel des constructeurs par défaut des attributs de la classe doit être pris en charge explicitement par le constructeur par défaut de l'utilisateur.

# Le constructeur par copie (clonage)



Le constructeur par copie est celui qui procède à l'initialisation d'un objet par copie d'un autre objet. Sa déclaration prend la forme suivante (une référence sur un objet de même classe et éventuellement une succession de paramètres initialisés par défaut) :

```
Complex( Complex &c, type val1=..., type val2=... );
```

ou :

```
Complex( const Complex &c, type val1=..., type val2=... );
```

Exemple :

```
class Complex {
private :
  double x, y;

public :
  Complex( double a=0., double b=0. ) { x = a; y = b; }
  Complex( const Complex &c ) { x = c.x; y = c.y; }
};
```

Le constructeur par copie est appelé dans l'initialisation suivante :

```
Complex C = Complex( 2., 2. );
```

S'il n'a pas été défini, le compilateur en synthétise un par défaut qui se contente de recopier les attributs d'un objet dans l'autre. **Il est également appelé lors du passage d'objets par valeur aux fonctions.**

# Le constructeur par copie (clonage)



Lorsqu'une classe dispose d'attributs pointeurs, le constructeur par copie généré par le compilateur peut poser problème car les adresses sont copiées, conduisant au partage de données par plusieurs objets.

# Construction des attributs



Lorsqu'une classe contient des objets, il convient de les construire correctement lors de la construction de la classe. Lorsque les objets membres n'ont pas de constructeur par défaut, il est possible de les initialiser en utilisant la syntaxe suivante :

```
Complex::Complex( double xx, double yy )
  : x (xx), y(yy) {
  ...;
  ...;
};
```

# Constructeur de conversion



Il s'agit d'un constructeur appelé avec un seul argument et dont le type de l'argument est celui d'un objet d'une autre classe.

# Destructeur



Le destructeur est une méthode qui porte le nom de la classe (comme le constructeur), mais précédé du caractère « tilda »